有关 VAE 的探索

1、 前言

可变编码器 VAE 在今天不光是在图像生成领域为各种 Diffuson Model 所用，也有作为文本翻译、语义通信相关内容的尝试。这里简单说一下我在 VAE 做的一些探索。

2、 从自编码器开始：

让我们先从自编码器开始，如下图所示，自编码器的网络像一个空竹一样，数据从输入层 x 输入进去后经过 Bottleneck 层 Z 后再输出 Y，如果只是简单的复制那中间的 Bottleneck 层也没啥用了，正因为有了 Bottleneck 层这样的窄层，这个网络就必须思考什么样的信息是最重要的信息，这也就带来了去噪、数据压缩等应用。

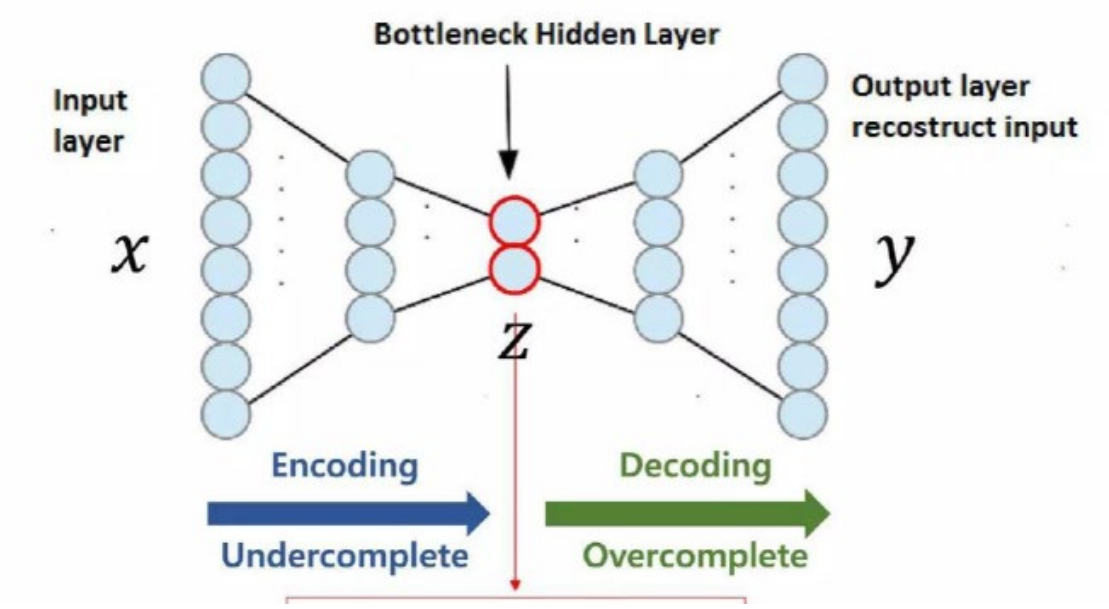

图 2-1：自编码器的架构

3、 从 AE 再到 VAE：简单概述

在机器学习中有一个很多人常提及的流形假设，即认为现实中的很多高维数据集实际上沿着高维空间内的低维潜流形分布。

基于此，很多 AutoEncoder 模式都是为了寻找这样的低维潜在的流形分布，如 DAE、CAE 等结构，但是这些模型的中间层是固定的参数，本身生成能力并

没有很强。VAE 采用贝叶斯公式，试图直接算出中间 Bottleneck 层的概率分布，也就是传统的贝叶斯公式：

$$P(Z|X)=\frac{P(X|Z)P(Z)}{P(X)} \quad (1)$$

看起来很美好，但是在深度学习中，这个 Z 是高维空间，很难计算，因此 VAE 提出能否算一个近似的答案，自此引入变分推断。

和传统方法企图能直接算出结果不同，VAE 先退一步，从一个常见的概率分布如正态分布开始，一步步调整直到与我们想要的 $P(Z|X)$ 接近。

为了能够让两个概率分布尽可能接近，VAE 引入了概率分布中常见的 KL 散度作为 Q 和 $P(Z|X)$ 两个概率分布的衡量，但这样看起来很怪，我都不知道 $P(Z|X)$ 是什么，我又怎么能算出来 KL 散度呢？从这里开始就引入 ELBO 也就是变分下界的概念了。

$$\text{ELBO}=\mathrm{E}_{\mathrm{Q}}(\log P(X|Z))+\mathrm{KL}(Q(Z|X)\|P(Z)) \quad (2)$$

公式 (2) 中的 KL 散度和上面不同，这是 Q 与先验 $P(Z)$ 的距离，而前面的 $\mathrm{E}_{\mathrm{Q}}(\log P(X|Z))$ 为重构误差，它可以是 BCE 或者 MSE，这里的 $P(Z)$ 是我们预先约定好的简单分布。利用对贝叶斯公式的推导，可知 ELBO 越大，Q 和 $P(Z|X)$ 两个概率分布的 KL 散度越小，这样损失函数的计算就从原先的计算 KL 散度变成计算 ELBO 就可以了。

看起来一切都很完美，只需要训练就行了？好像并不是，我们都知道在神经网络的训练中参数的调整需要依靠反向传播，但是从 Z 中采样 z 是随机采样，随机采样是不可导的，也就没法进行反向传播，训练就没法进行了。为了能够训练，论文提出“重参数化技巧”，也就是改变 z 的采样方式，使其变得可导：

$$z=\mu+\sigma\times\varepsilon \quad (3)$$

其中，ε 服从 (0,1) 正态分布，μ 和 σ 分别为均值和方差，通过这种方式计算的 z 就变得可导了，训练则是采用小批量训练不断循环和更新参数。

4、 从 MNIST 开始复现

VAE 文献就是从 MNIST 数据集开始训练的，外加 MNIST 数据集都是灰度照片，训练难度相对较低，我们也将从 MNIST 开始复现。

硬件环境：CPU 为 Intel Xeon CascadeLake 8vCore，56GB 内存，GPU 为一张 NVIDIA Tesla T4（不过后来发现笔记本基本也能跑）。

框架采用 PyTorch 2.8.0，Pyhton 版本为 3.12.1，VAE 代码如下：

```
class VAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        # input_dim: 输入维度
        # hidden_dim: 隐藏层维度
        # latent_dim: 潜在变量维度
        super(VAE, self).__init__()
        # 编码器
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        # 均值和对数方差的全连接层
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)      # 均值层
        self.fc_logvar = nn.Linear(hidden_dim, latent_dim) # 对数方
        # 解码器
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, input_dim),
            nn.Sigmoid()
        )
```

由于 MNIST 是二值化的图像，因此在损失函数中重构损失部分采用的是 BCE。

```
def vae_loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return BCE + KLD, BCE, KLD
```

训练跑了 10 轮，隐藏层维度为 400，中间层维度为 20，优化器使用的是 Adam，训练过程中损失变化如下图：

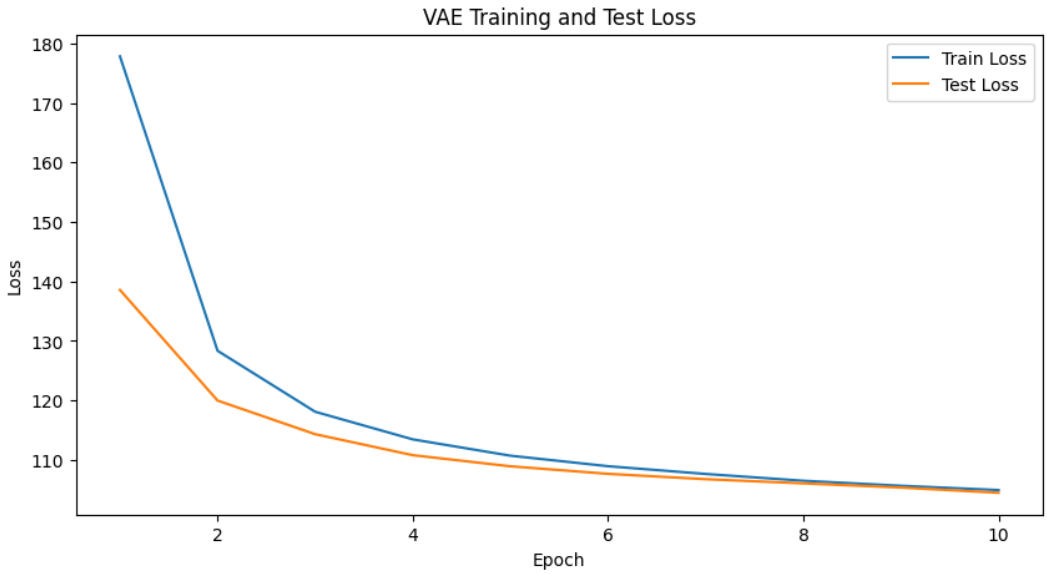

图 4-1： MNIST 数据集下训练的损失变化

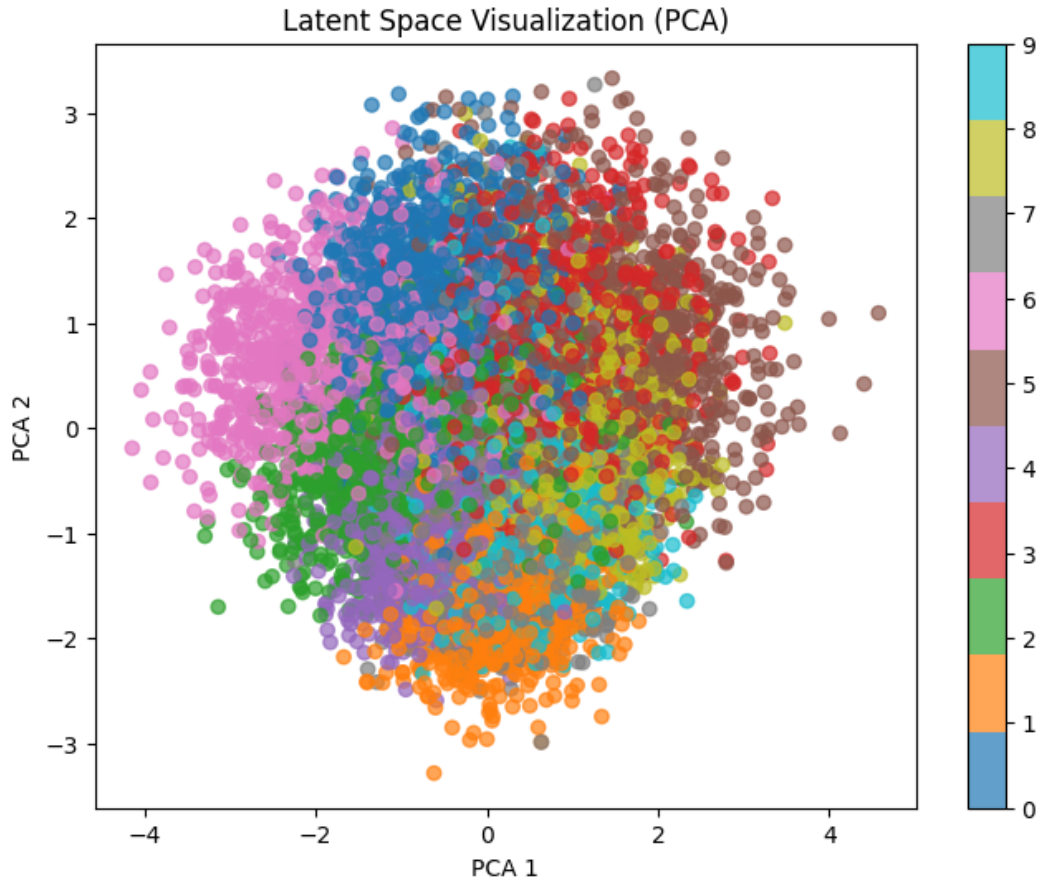

图 4-2：模型的 PCA 主成分分析

从 PCA 主成分分析的结果来看，每一类的生成的都还算集中，那么实际生成的效果如何呢？生成的图片如下：

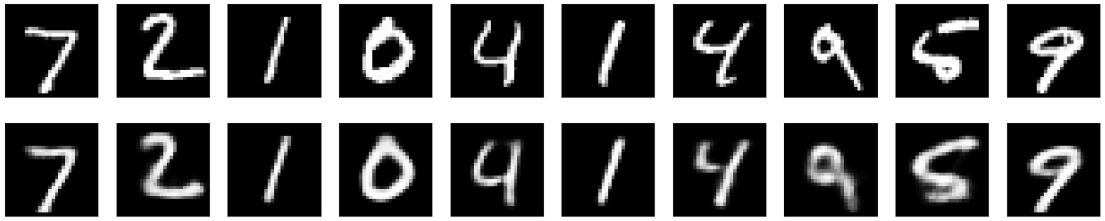

图 4-3：上图为数据集样本，下图为生成的样本

在这个参数的基础上尝试分别减少隐藏层数量到 200 、减少中间层数量到 10，生成的图片如下：

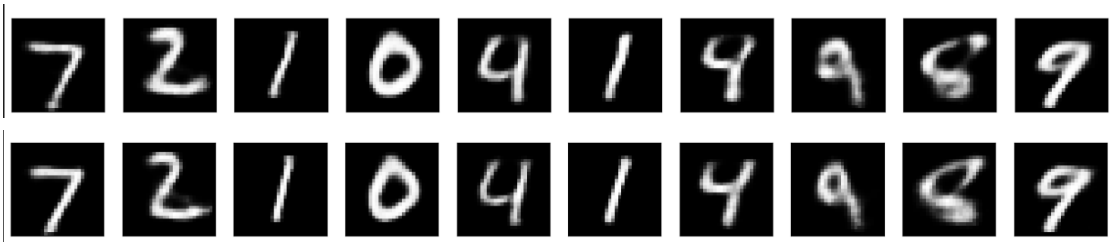

图 4-4：上图为隐藏层数量 200 的样本，下图为中间层数量 10 的样本

直接看效果的话，对比原先没减少参数的模型损失的效果是不明显的。这可能跟数据集本身是小尺寸灰度图像有关，因为小尺寸的灰度图像意味着需要学习的特征并没有那么多。

表 4-1：不同参数模型的评估数据

	ELBO	MLE	MSE
Model 1	-104.675	-98.331	0.0163
Model 2	-118.023	-106.975	0.0224
Model 3	-113.538	-104.684	0.0207

表中，Model 1 为没减少参数的模型，Model 2 和 Model 3 分别为减少隐藏层和中间层的模型，MLE 列为边际似然估计，使用重要性采样的方法。

从表中数据来看，对比模型 1，模型 2 和 3 在减少隐藏层和中间层的情况下，ELBO、MSE、MLE 三个指标均有所上升但并不明显，意味着小尺寸灰度图像下并不需要大量的中间层或者隐藏层。

5、 从灰度图像到彩色，从全连接神经网络到卷积神经网络

更换数据集到 CIFAR10，由于数据集从灰度图像变成彩色图像，损失函数也要进行修改，将重构损失函数从原先的适用于数据为 [0,1] 的二元交叉熵换成 MSE。

具体的代码如下：

```
def vae_loss_function(recon_x, x, mu, logvar):
    MSE = F.mse_loss(recon_x, x, reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return MSE + KLD
```

看起来一切都很美好？那么直接跑起来看看？

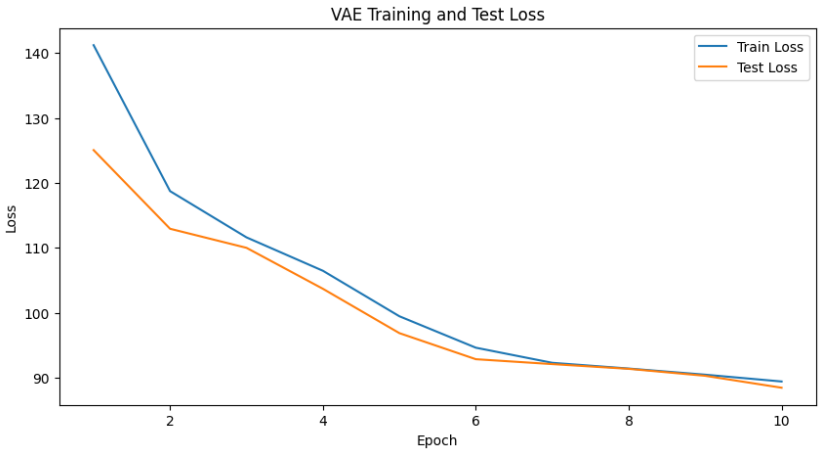

图 5-1：CIFAR-10 数据集全神经网络训练的损失变化

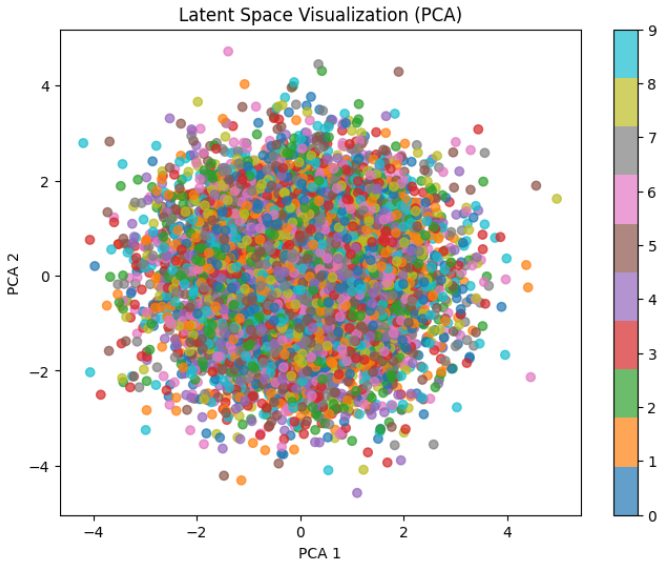

图 5-2：上述模型的 PCA 主成分分析

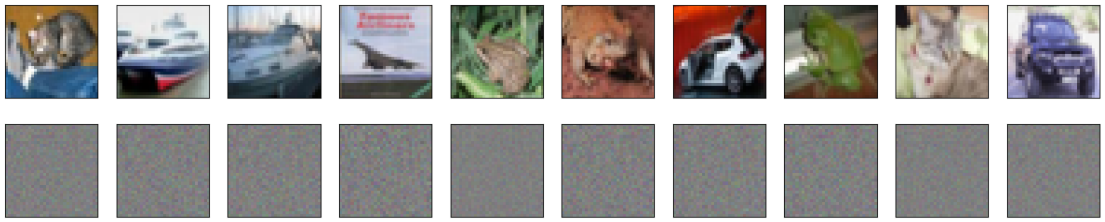

图 5-2：上图为数据集样本，下图为生成的样本

怎么全是灰色的？那是因为我们并没有修改之前代码的参数，使得模型仍然按照 MNIST 中的隐藏层和中间层数量去训练，训练的过程中不光会丢掉无效的信息，为了塞进低维空间也不得不丢掉大量的有效信息，最终导致生成的图片如电视雪花屏一样的效果。

那么我们把中间层调整至 256 呢？效果确实会好一些：

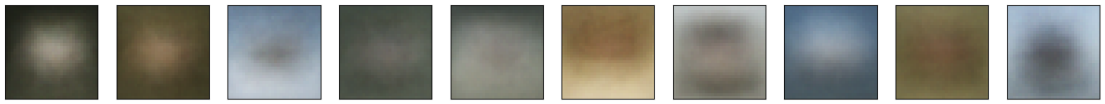

图 5-3：增大中间层数量的效果

有所改进，但是还是很模糊，我们还可以尝试增加层数量至 2048 使得训练集数据传入网络后能学到更多的特征，这样的效果如下：

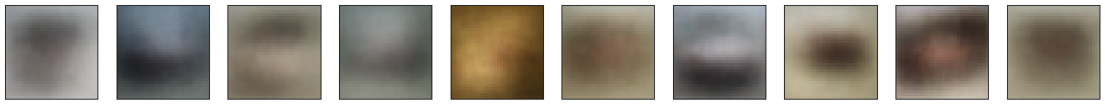

图 5-4：增加隐藏层数量的效果

看起来效果还是不好，我们需要重新审视一下网络，目前将图像数据传入网络时是将二维的图像展平成一维数据再传入网络进行训练的，这样会丢掉像素与像素之前的联系，在原先灰度数据时丢掉像素之间的联系影响可能还没有这么大，但是在二维的图像下这就会让网络缺少大量可以学习和压缩的信息。如何改进呢？我们可以把全连接的网络改成卷积神经网络，下面是编码器的部分，解码器同理。

```
class ConvVAE(nn.Module):
    def __init__(self, latent_dim):
        self.encoder = nn.Sequential(
            nn.Conv2d(3,32,kernel_size=3,stride=2, padding=1),
            nn.BatchNorm2d(32), # 批量归一化
            nn.ReLU(),
            nn.Conv2d(32,64,kernel_size=3,stride=2,padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64,128,kernel_size=3,stride=2,padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128,256,kernel_size=3,stride=2,padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Flatten(),
        )     # 编码器 - 使用卷积层
```

效果如何呢？图片的轮廓能看的更清楚了些，但还是很模糊，但 ELBO 从原先的-92.6265 提升至-81.5941。

图 5-4：更换卷积神经网络的效果

此时的损失曲线如下：

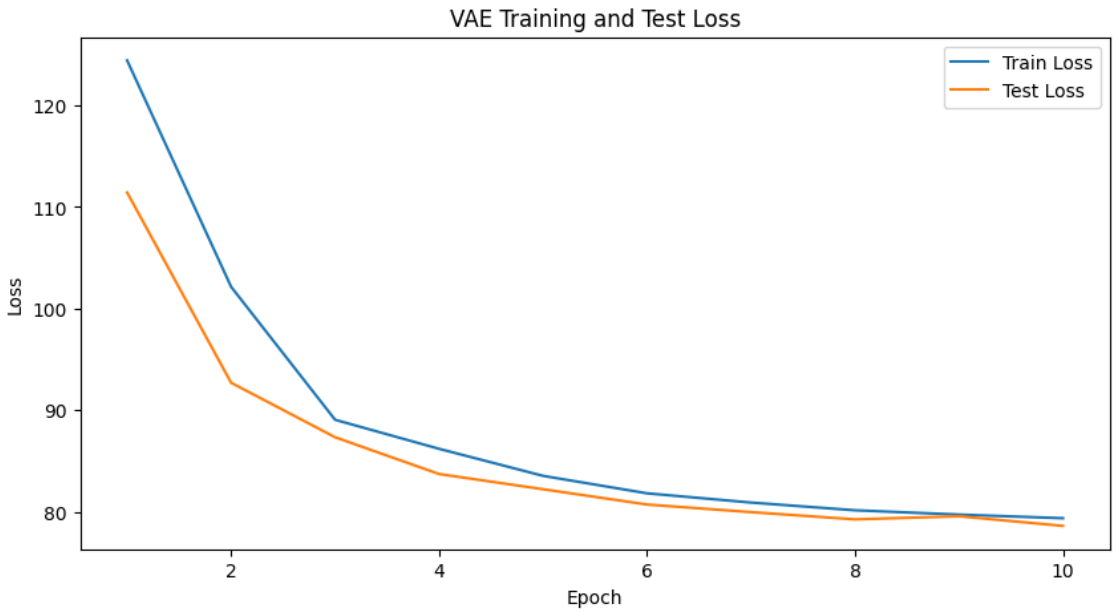

图 5-5：CIFAR-10 数据集卷积神经网络训练的损失变化

还有什么别的方式可以优化吗？有的，我们可以更换激活函数，从 ReLU 函数换成 LeakyReLU 函数，使得在负输入的时候仍然能提供一定的梯度，我们也可以在损失函数中为重构损失的 MSE 函数提供负输入的梯度计算。

$$\text{LeakyReLU}(x)=\begin{cases} x, \text{ if } x \geqslant 0 \\ \text{negative\_slope} \times x \text{ other} \end{cases} \quad (4)$$

我们令负输入时的斜率为 0.2，最终生成的效果如下：

图 5-6：优化损失函数并更换激活函数的效果

6、 总结与展望

CIFAR-10 的训练到这里我已经怀疑人生了，后来去 Github 和 arxiV 上翻了一下其他人的训练成果，要么就是画饼要么也是就这个样，其实还可以试试 VQ-VAE 这种从连续分布走向离散分布的 VAE 架构，后面再试试吧。

参考文献

[1] Kingma, Diederik P., and Max Welling. "Auto-encoding variational bayes." arXiv preprint arXiv:1312.6114 (2013).

[2] Doersch, Carl. "Tutorial on variational autoencoders." arXiv preprint arXiv:1606.05908 (2016).

[3] Semeniuta, Stanislau, Aliaksei Severyn, and Erhardt Barth. "A hybrid convolutional variational autoencoder for text generation." arXiv preprint arXiv:1702.02390 (2017).